ECE 6276 DSP Hardware System Design (Fall 2018)

Project Report

Floating Point Arithmetic

Team Number: 3

Dingxin Jin	gtID 903429811
Shengwei Lyu	gtID 903430584
Peng Guo	gtID 903424176
Xiaoting Lai	gtID 903410194

Date: 12/03/2018

Table of Contents

1 Introduction

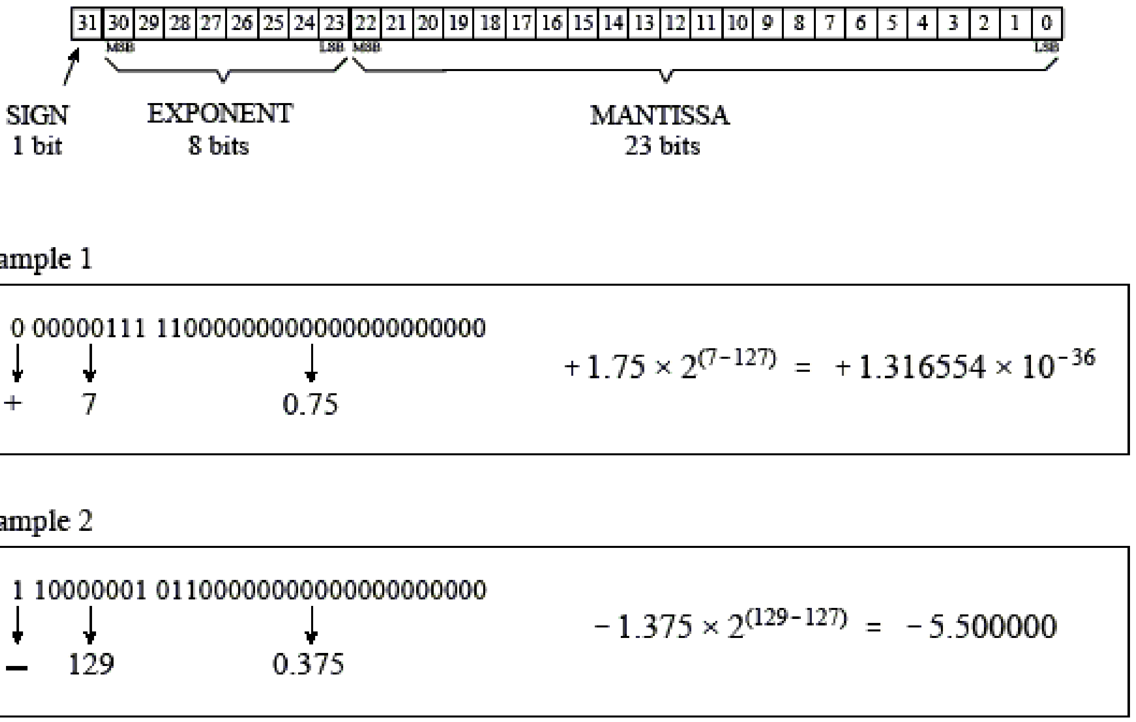


Figure 1 Floating Point Arithmetic Introduction

The floating-point representation for 32-bit is to separate numbers in three parts, the sign bit is the leading bit, the exponent includes the following 8 bits, and the rest 23 bits are mantissa. The range of the exponent is -126 to 127, transferring the largest number back to decimal number, 2^127 × (2 − 2−23) ≈ 3.4028234664 × 10^38, so it enlarges the range that a 32-bit number is able to represent, the tradeoff is the precision, which is 24 bits.

For example, shown in Figure 1, the first number has sign bit 0, representing positive number, the exponent part is 7, and the actual binary exponent is achieved by subtracting the bias, 127. Finally, adding 1 to the mantissa and multiplying the result by 2 to the -120 to get the decimal representation.

This floating-point arithmetic includes all four operations, the adder and subtractor are built in one cluster, multiplier and divider each takes one cluster. The multiplier and divider also share one normalization block.

2 Features

This floating-point arithmetic handles all types of single precision input, including special cases, such as infinity and NaN, and correctly handles denormalized number. The test cases include over 100 thousand input data, and all arithmetic handle them correctly and quickly.

3 Target Specifications

Table 1 Target Specifications

Number Type	Word Length	Mantissa	Exponent	Bias
Floating-point	32 bits	23 bits	8 bits	127
Input / Output Way	Target Frequency	Sampling Period	Throughput	Range
UART	100M	20	9600 bit/s	2^128

4 Design Architecture

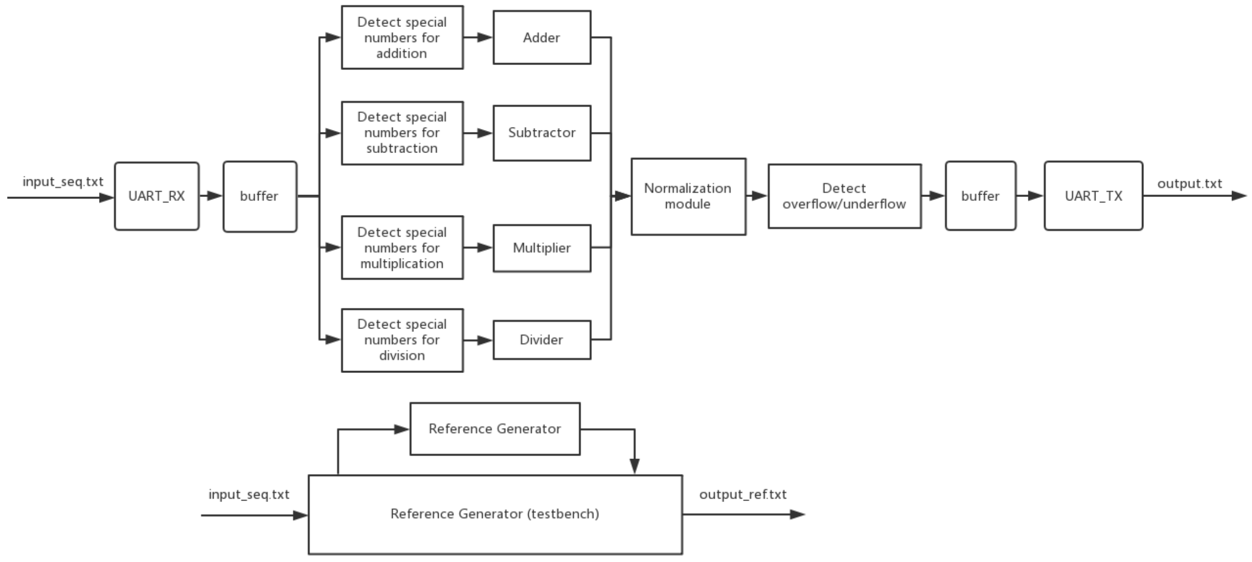

input_seq.txt
UART_RX
buffer
Detect special numbers for addition
Adder
Detect special numbers for subtraction
Subtractor
Detect special numbers for multiplication
Multiplier
Detect special numbers for division
Divider
Normalization module
Detect overflow/underflow
buffer
UART_TX
output.txt
Reference Generator
input_seq.txt
Reference Generator (testbench)
output_ref.txt

Figure 2 Overall Architecture

All the data and operands are inputted through UART from computer. The buffer on the input side arrange the data into good manner then put it into functional module, which is integrated as Wrapper.vhd. The input data includes valid bit and control bits to show that whether this data is valid and which arithmetic operation should this data undergo *(00 for *, 01 for /, 10 for +)*. We use adder to do the subtraction for the reason that we just need to invert the sign bit for data and subtraction can be made. Another thing to mention is that **all functional module can do calculation with denormalized number, and check all kinds of exceptions.**

After calculation, we use normalization module to normalize our results and check if there is overflow or underflow. Notice that multiplier and divider use the same normalization module, and adder use its own normalization module. Then we use another buffer to put the result into UART and check sanity on PC.

The golden reference is generated by Python and floating-point library in VHDL. **Two types of reference can be generated.** One is used to simulate pipeline (collect valid/invalid results every cycle), and the other one is to simulate state machine (only collect valid results).

4.1 uArch 1: UART (UART_RX.vhd UART_TX.vhd TOP.vhd)

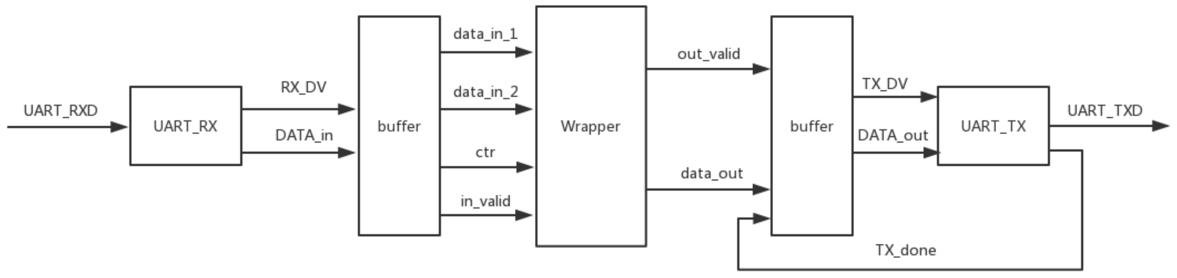


Figure 3 uArch 1: TOP module

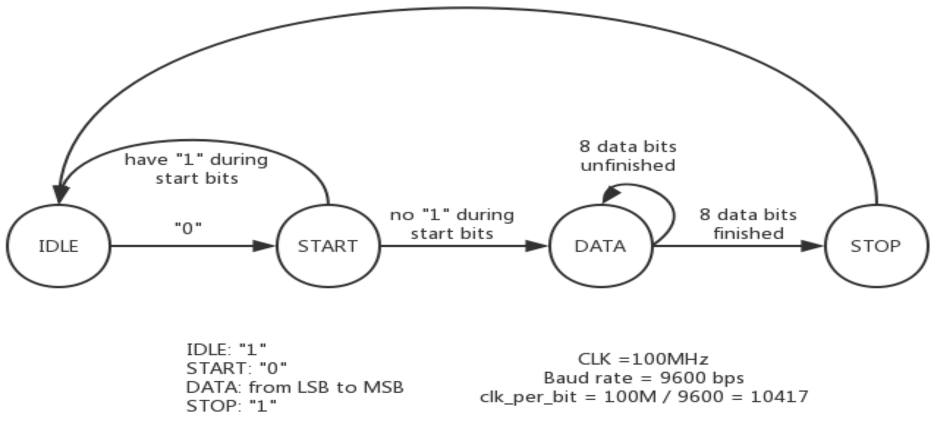


Figure 4 uArch 1: UART

The UART module and the buffers receive serial data transmitted from the USB port, transform the serial data into 2 32-bit floating point numbers, 1 valid bit and 2 control bits (to choose which kind of operation will be performed), then feed the data into the Wrapper module to finish the floating-point calculation. After the Wrapper module produce a valid output data, transform the answer (32-bit floating point number) into serial bits and transmit the bits out from the USB port.

Serial data received from the USB port (UART_RXD) is stored as std_logic. When 1 UART transmission finished, the RX data_valid (RX_DV, std_logic) will be set as 1 to indicate that now the 8-bit data (DATA_in, std_logic_vector) can be fetched. Because one 32-bit floating point number need 4 times of transmission and the valid and control bits need 1 transmission, we need 9 UART transmissions before doing the calculation. In the input buffer, we use a counter (4-bit std_logic_vector) to determine which part of the data are we receiving, then organize the data as data_in_1(32-bit std_logic_vector), data_in_2(32-bit std_logic_vector), ctr(2-bit unsigned), and in_valid (std_logic).

After the calculation, the Wrapper module will produce an out_valid signal (std_logic) and a 32-bit data_out (std_logic_vector). In the output buffer, we use the out_valid and tx_done(indication the end of a TX transmission, std_logic) signals to generate another counter (4-bit std_logic_vector) to split the data_out into 4 8-bit data and send them into the TX module, then output the serial data UART_TXD (std_logic) from the USB port.

Inside the UART TX and RX modules, we use state machines to describe the behaviors. The modules are initially in an idle state. When a transmission is needed, first send the start bit (‘0’), this is the start-bit state. Then send the 8-bit data from LSB to MSB (data_bits state), finally send the stop bit ‘1’ (stop-bit state), set the tx_done signal to 1 for 2 clock cycles and the machine returns to idle state.

4.2 uArch 2: Floating Point Adder (adder.vhd)

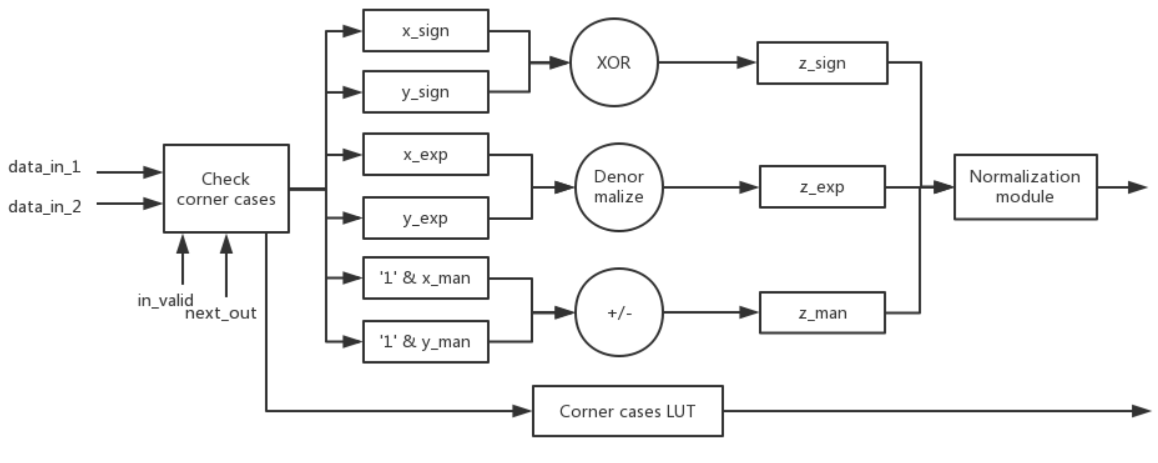


Figure 5 uArch 2: Floating Point Adder

The Floating-Point Adder implements addition on valid input data. The input and output are 32-bit precision, also rounding is included. The calculation is based on the basic math principle, and also includes all exception cases.

The floating-point number is stored as std_logic_vector format, and it consists of three separate parts--the sign (the leading bit), the exponential part (the 31st bit down to the 24th bit), and the mantissa part (the rest 23 bits at the end).

First, the input data should be valid with the in_valid signal, the adder should also check the next_out signal to determine whether it is able to output the calculated result. If the input data is valid, then check the exception cases of nan (not a number). Second, for the sign bit, use XOR logic to determine the input data should be added or subtracted. And compare the exponent and mantissa to get the sign bit of result. For the exponent part, denormalize the input data with the selected larger exponent. Then, implement the addition or subtraction with the denormalized mantissa part to get the initial sum. Besides, mark the underflow and overflow cases with flag bits. Finally, send the three separate results to the normalization module.

4.3 uArch 3: Floating Point Divider (fpp_divide.vhd)

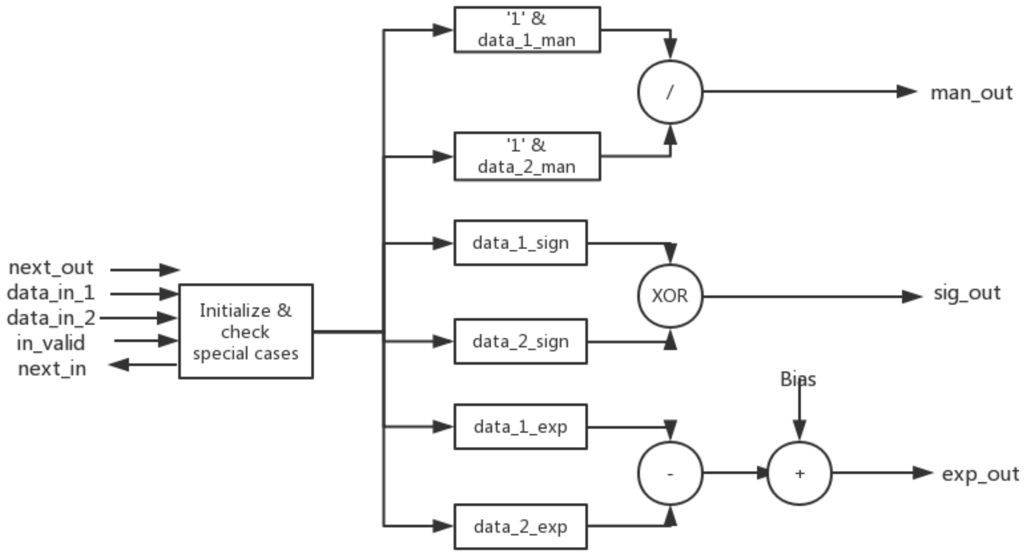


Figure 6 uArch 3: Floating Point Divider

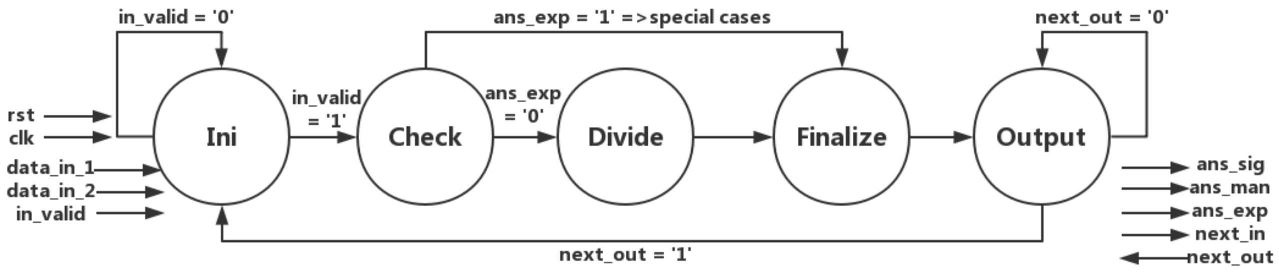


Figure 7 Floating Point Divider State Machine

The Floating-Point Divider implements division on valid input data. The input and output are 32-bit precision, also rounding is included.

Every floating-point number is stored in three separate parts--the sign (the leading bit), the exponential part (the 31st bit downto to the 24th bit), and the mantissa part (the rest 23 bits at the end). Because the sign bit of the result can be simply achieved by doing exclusive or on the input sign bits, it is stored in std_logic type. The exponent and mantissa parts, as they all require more complex arithmetic, are stored as unsigned type.

After inputting valid data, the first thing done is to check if any data is denormalized. The normalization is done in 4 steps: First, reverse the bit order of the data so that the leading one becomes the last one from the end. Second, decrement the result from step one. Third, do XOR on the result from step one and step two. Last, and the result of step three and step one, leaving there only one ‘1’ in the line to indicate the location of the leading one. If the ‘1’ is the third bit from the end, meaning the leading one of the original data is the third bit from the beginning (details of normalization refers to 4.6). If both inputs are normalized, the block checks for exception cases which are listed in “Exception_Definition.xls”. The exponent part is subtracting the exponent of the dividend by the exponent of the divisor. The mantissa part is done by first shift both dividend (24 bits with ‘1’) and divisor (24 bits with ‘1’) by 24 bits. For every cycle in the total of 26 cycles, the block compares the remainder and the divisor, if the remainder is larger, the quotient gets ‘1’ at the end, otherwise gets ‘0’, and for every cycle, the divisor is shift one bit to the right, the quotient shift one bit to the left, done before filing ‘0’ or ‘1’ at the end. When all 26 cycles are done or the remainder is zero, the mantissa part complete.

4.4 uArch 4: Floating Point Multiplier (FP_mult.vhd)

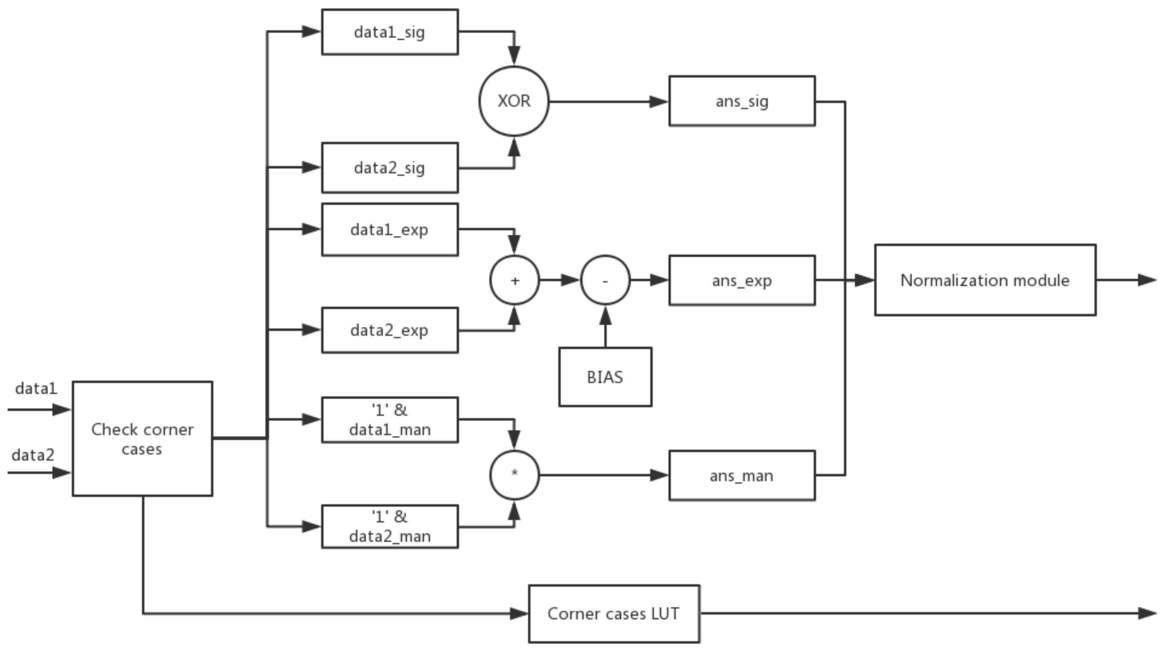


Figure 8 uArch 4: Floating Point Multiplier

The Floating-Point Multiplier implements multiply on valid input data. The input and output are 32-bit precision, also rounding is included. However, the mantissa of the output is 48 bits because it is 24 bits multiply by 24 bits.

Every floating-point number is stored in three separate parts--the sign (the leading bit), the exponential part (the 31st bit downto to the 24th bit), and the mantissa part (the rest 23 bits at the end). Because the sign bit of the result can be simply achieved by doing exclusive or on the input sign bits, it is stored in std_logic type. The exponent and mantissa parts, as they all require more complex arithmetic, are stored as unsigned type.

After inputting valid data, the first thing done is to check if any data is denormalized (details of normalization refers to 4.6). If both inputs are normalized, the block checks for exception cases which are listed in "Exception_Definition.xls". The exponent part is adding the exponent of the dividend by the exponent of the divisor. The mantissa part is done by multiplying both mantissa parts of the inputs. For cases of carry one, the block checks for the leading bit, bit (47), of the mantissa, and if bit (47) is '1', increments the exponent.

4.5 uArch 5: Normalization (Normalization.vhd)

We try to come up with a normalizer which can calculate out the result with a sample period of 1.

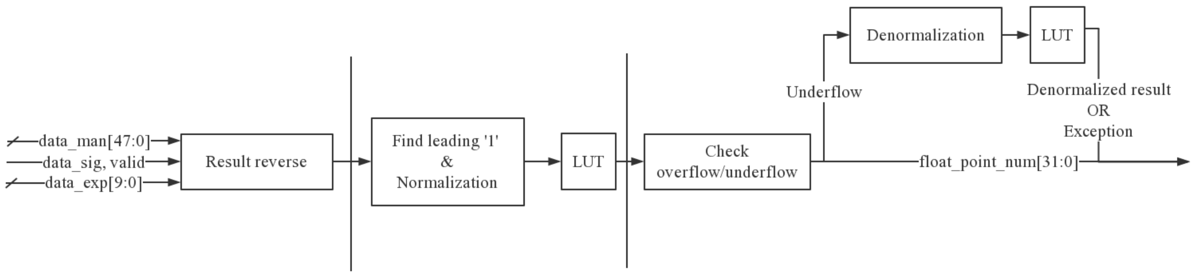


Figure 9 uArch 5: Normalization & Denormalization part

Algorithm for Normalization: (e.g. find leading '1' for b:00101001)

(1) Result reverse: [MSB to LSB] b:00101001 → d:10010100 [LSB to MSB]

(2) Find leading '1':

A. Decrement: d: 10010100 – 1 = d': 10010011

B. XOR d with d': 10010100 xor 10010011 = x :00000111

C. AND d with x: 10010100 and 00000111 = x':00000100

Now we have an 'one-hot' code to show us the location of leading '1', we can use it as an index for LUT to give us the distance for shifting.

(1) Check if underflow happens (“01 & exp” for overflow, “11 & exp” for underflow)

(2) Calculate the distance d_Exp from exp to “00000000”.

(3) Check if underflow result can be recovered by denormalization

```
(d_Exp ≤ 23)
```

(4) Give the denormalized result using d_Exp as an index for a second LUT

5 Implementation Battles and Solutions

5.1 Rounding and Borrow ‘1’ from Exponent for Divider

When the dividend is smaller than the divisor, it requires to borrow ‘1’ from the divider. After getting the 26 bits mantissa (25 downto 0) of the result, it is divided into 2 cases. First, the first bit is ‘1’, this is when the dividend is larger or equal to the divisor, the mantissa part of the result is bit (24 downto 2), the rounding considers the nearest bit, bit (1), if bit (1) is ‘1’, the mantissa of the result increment by 1. The second case is then the first bit is ‘0’, when the dividend is smaller than the divisor, in this case, the result takes bit (23 downto 1) and consider bit (0) for rounding. For some of the special cases, such as overflow and underflow where rounding is unnecessary, the rounding implementation then excludes those cases.

5.2 Overflow and Underflow Judgement for Adder

When the result is too small or too large to be denoted by the 32-bit floating point format, especially with the 8-bit exponent, the flag bits of overflow or underflow should be marked.

As for adder, if the most significant bit of the initial sum is 1, then consider whether the result is overflow or not. If the most significant bit of the initial sum is 0, then consider whether the result is underflow or not. The result is underflow if the difference between the larger exponent and leading 1 position is not larger than 0. And the result is overflow if the sum of the larger exponent and the leading 1 position is not smaller than 255.

5.3 Timing and Verification Issue for the Whole System

The attempt for doing normalization and denormalization in 1 clock cycle brings a lot of timing problems. Firstly, we try to use huge bunch of combinational logic and result in a WNS of -200ns. So, we tried to reorder our code, break critical path with pipeline structure, and transform large loop into few small loops. As a result, the WNS is increased to +1.524ns.

As for division part, we naively tried do division in one clock cycle at the very beginning, and this also make our system cannot meet the timing. The solution is to implement divider with state machine and do the division with multi cycles. The result shows that both area and timing for divider become better, while the harm is that we have to build another testbench and reference to test the functionality of state-machine based modules.

6 Testbench Details and Simulation Results

6.1 State-Machine Based Testbench (tb_adder.vhd)

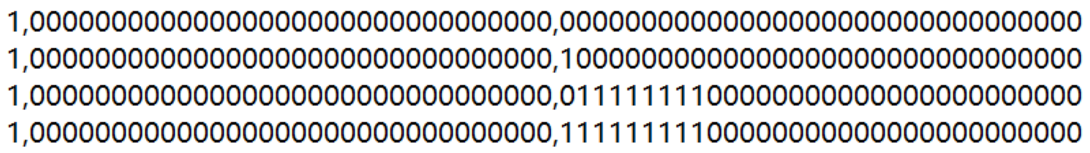
Both divider and adder are implemented with state machine. Let's cite tb_adder.vhd as an example. (P.s. divider need to be tested together with normalization module)

This testbench verifies the result coming from the adder cluster. By using 100 thousand lines of input, it is able to test edge cases.

6.1.1 Input/Output File Format

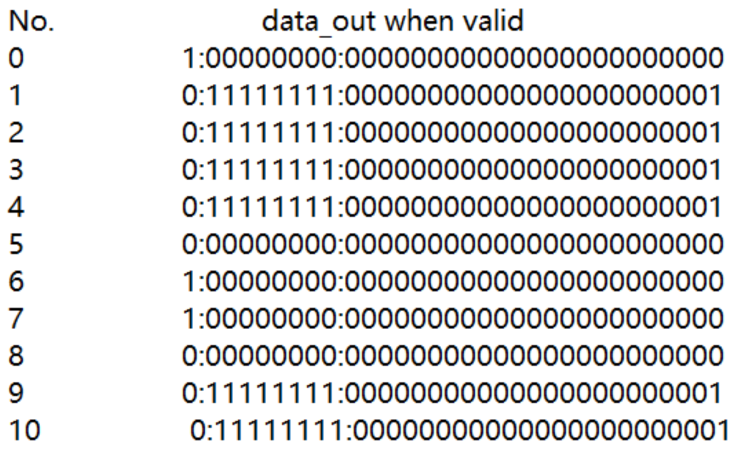
Input and output are all text files.

For input side, first column is valid bit, the remaining 2 columns are two operands. We also have next_out bit saved in input_out_ready.txt for each input to test handshake function.



```
1,00000000000000000000000000000000,00000000000000000000000000000000
1,00000000000000000000000000000000,10000000000000000000000000000000
1,00000000000000000000000000000000,01111111000000000000000000000000
1,00000000000000000000000000000000,11111111000000000000000000000000
```

For output side, the first column is the count of valid input. The second column is the output 32-bit result. Separated by column are the sign bit, exponent, and mantissa. Notice that only valid outputs are collected.



```
No.                data_out when valid
0              1:00000000:00000000000000000000000
1              0:11111111:00000000000000000000001
2              0:11111111:00000000000000000000001
3              0:11111111:00000000000000000000001
4              0:11111111:00000000000000000000001
5              0:00000000:00000000000000000000000
6              1:00000000:00000000000000000000000
7              1:00000000:00000000000000000000000
8              0:00000000:00000000000000000000000
9              0:11111111:00000000000000000000001
10              0:11111111:00000000000000000000001
```

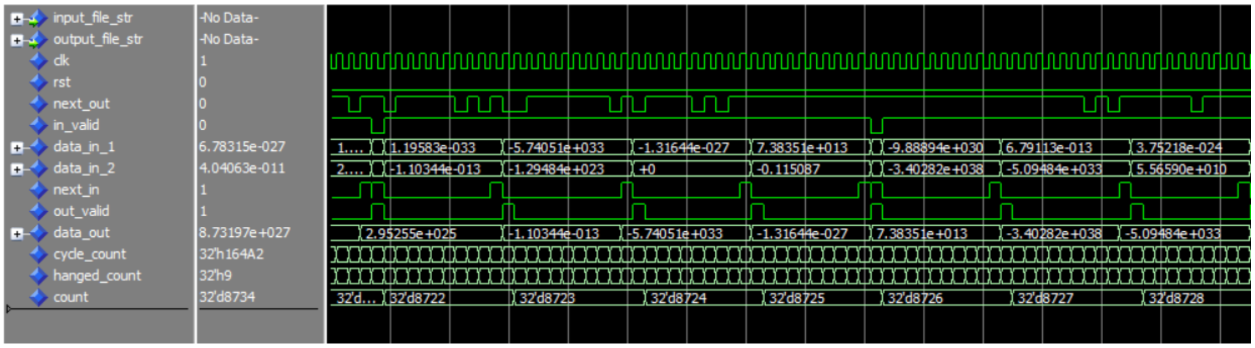
6.1.2 Patterns

If in_valid is low, the testbench generates an invalid output. If next_in is low, it requires the testbench to stall the current data for another cycle. If next_out is low, the testbench cannot take in data in the current cycle, and the testbench will consider the output for this cycle as not valid so that it will not collect the result.

6.1.3 Simulation Results

The text result is shown above, and the figure below shows the waveform. Noticed that out_valid and next_in almost have the same interval because the state-machine based adder/divider may take multi cycles to process the data, while the maximum speed for testbench to feed input data is per clock cycle, which is much faster and so that keep the adder/divider busy for processing data

Also, the count signal is the ‘No.’ showed in the text result above, which is used to locate the data and make debug easier. We cannot use cycle_count (clock cycle) here for clock cycle is meaningless in these state-machine cases.



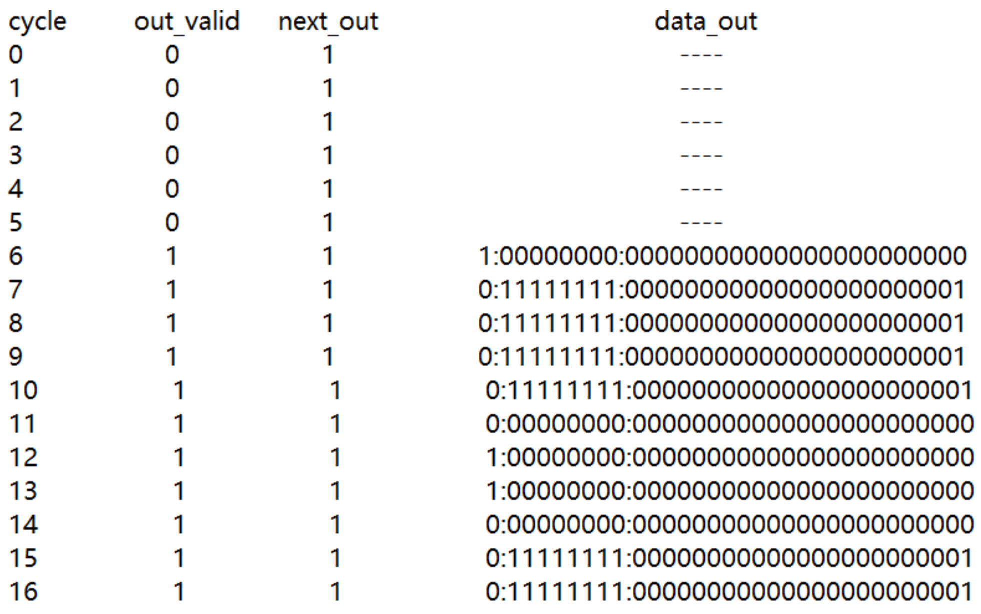
```
input_file_str    -No Data-
output_file_str   -No Data-
clk               1
rst               0
next_out          0
in_valid          0
data_in_1         6.78315e-027
data_in_2         4.04063e-011
next_in           1
out_valid         1
data_out          8.73197e+027
cycle_count       32'h164A2
hanged_count      32'h9
count             32'd8734
```

Figure 10 TB1: Adder testbench simulation waveform

6.2 Pipeline Based Testbench (tb_Multplier_Wrapper.vhd)

This testbench verifies the result coming from the multiplier (fully pipelined) & normalization module. By using 100 thousand lines of input, it is able to test edge cases.

6.2.1 Input/Output File Format



```
cycle    out_valid   next_out                  data_out
0            0           1                       ----
1            0           1                       ----
2            0           1                       ----
3            0           1                       ----
4            0           1                       ----
5            0           1                       ----
6            1           1          1:00000000:00000000000000000000000
7            1           1          0:11111111:00000000000000000000001
8            1           1          0:11111111:00000000000000000000001
9            1           1          0:11111111:00000000000000000000001
10           1           1          0:11111111:00000000000000000000001
11           1           1          0:00000000:00000000000000000000000
12           1           1          1:00000000:00000000000000000000000
13           1           1          1:00000000:00000000000000000000000
14           1           1          0:00000000:00000000000000000000000
15           1           1          0:11111111:00000000000000000000001
16           1           1          0:11111111:00000000000000000000001
```

Figure 11 TB2: Multiplier testbench simulation output

Input and output are all text files.

For input side, it is the same with what is mentioned above.

For output side, we collect data every clock cycle, and the invalid data is showed as '----' in the figure. out_valid and next_out signal are also printed for checking the correctness of pipeline.

6.2.2 Patterns

If in_valid is low, the testbench generates an invalid output. If next_in is low, it requires the testbench to stall the current data for another cycle. If next_out is low, the testbench cannot take in data in the current cycle, and the testbench will consider the output for this cycle as not valid . However, it will still collect the result and print it out in the text file.

6.2.3 Simulation Results

The text result is shown above, and the figure below shows the waveform. Noticed that every input data only takes one cycle (if module available and input is valid) to be put into multiplier. Also, out_valid and next_in is usually high because the high throughput of pipeline.

Also, the cycle_count (clock cycle) signal is the 'cycle' shown in the text result above, which is used to locate the data and make debug easier.

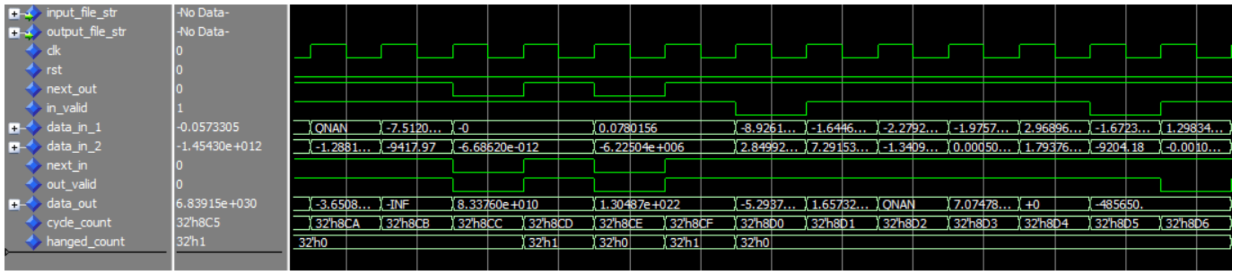


Figure 12 TB3: Multiplier testbench simulation waveform

6.3 System Level Testbench (tb_Wrapper.vhd)

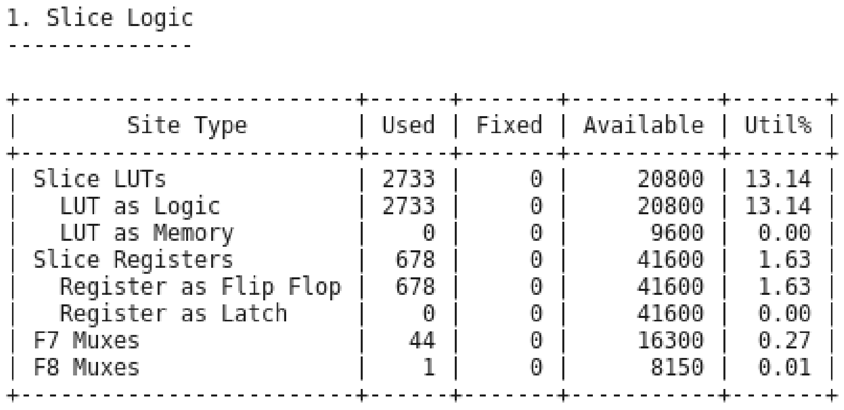
For the reason that only multiplier use pipeline structure, it is not necessary to test the whole system using pipeline based testbench. Instead, we will use the state machine based testbench to test our whole system.

The format, patterns, and simulation results are similar to what has been discussed in 6.1. The only difference is that we use two control bits to choose which arithmetic we want to test (00 for *, 01 for /, 10 for +). For convenience of testing, we set control bits a constant value when doing simulation, which means each arithmetic module is tested one-by-one when connected with other modules.

7 Implementation Results

Implementation results, including area and timing (WNS, WHS), for each uarch (no need for UART) as well as the system level.

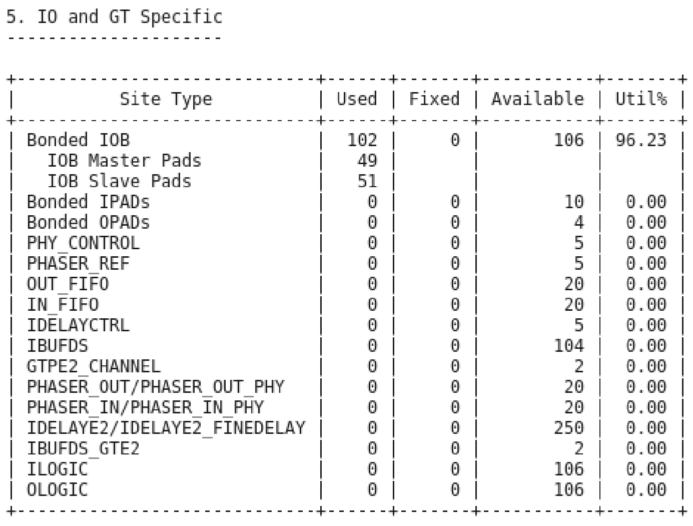
7.1 uArch 2: Floating Point Adder (adder.vhd)



```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              | 2733 |     0 |     20800 | 13.14 |
|   LUT as Logic          | 2733 |     0 |     20800 | 13.14 |
|   LUT as Memory         |    0 |     0 |      9600 |  0.00 |
| Slice Registers         |  678 |     0 |     41600 |  1.63 |
|   Register as Flip Flop |  678 |     0 |     41600 |  1.63 |
|   Register as Latch     |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                |   44 |     0 |     16300 |  0.27 |
| F8 Muxes                |    1 |     0 |      8150 |  0.01 |
+-------------------------+------+-------+-----------+-------+
```

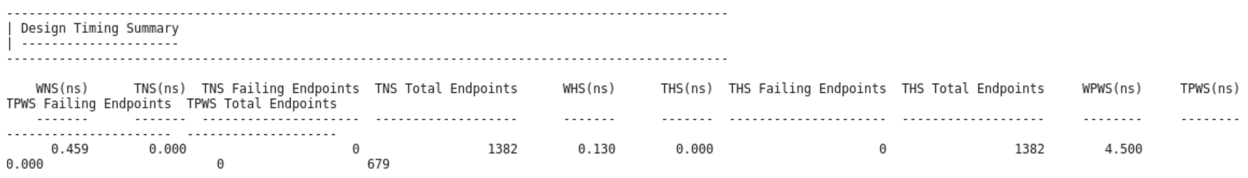
Figure 13 Multiplier Slice Logic



```
5. IO and GT Specific
--------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |  102 |     0 |       106 | 96.23 |
|   IOB Master Pads           |   49 |       |           |       |
|   IOB Slave Pads            |   51 |       |           |       |
| Bonded IPADs                |    0 |     0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |         5 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |       106 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

Figure 14 Multiplier IO

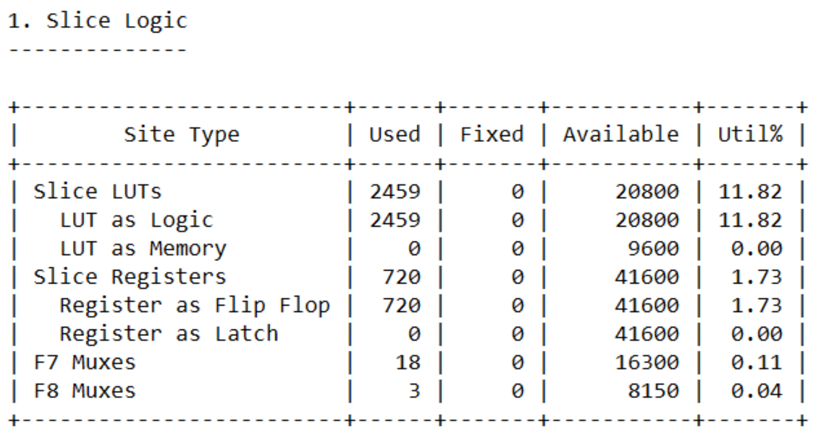


```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)
TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------  -------------------     --------     --------
----------------------  --------------------
      0.459        0.000                      0                 1382        0.130        0.000                      0                 1382        4.500
0.000                      0                  679
```

Figure 15 Multiplier Timing

7.2 uArch 3: Floating Point Divider (fpp_divide.vhd)

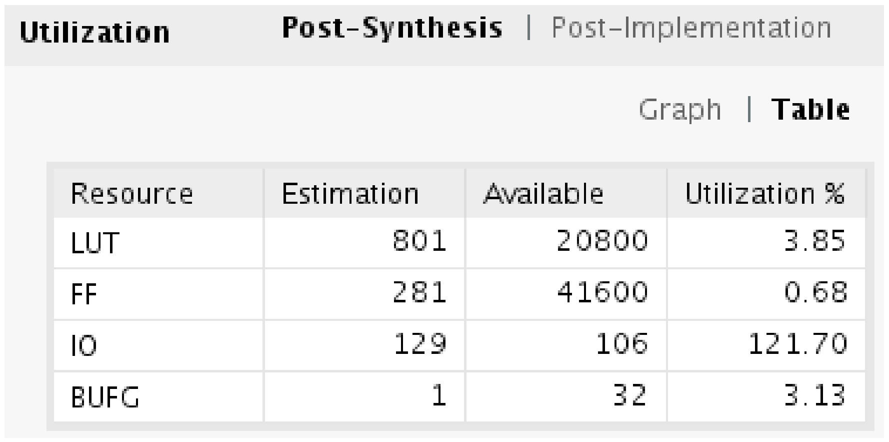


```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              | 2459 |     0 |     20800 | 11.82 |
|   LUT as Logic          | 2459 |     0 |     20800 | 11.82 |
|   LUT as Memory         |    0 |     0 |      9600 |  0.00 |
| Slice Registers         |  720 |     0 |     41600 |  1.73 |
|   Register as Flip Flop |  720 |     0 |     41600 |  1.73 |
|   Register as Latch     |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                |   18 |     0 |     16300 |  0.11 |
| F8 Muxes                |    3 |     0 |      8150 |  0.04 |
+-------------------------+------+-------+-----------+-------+
```

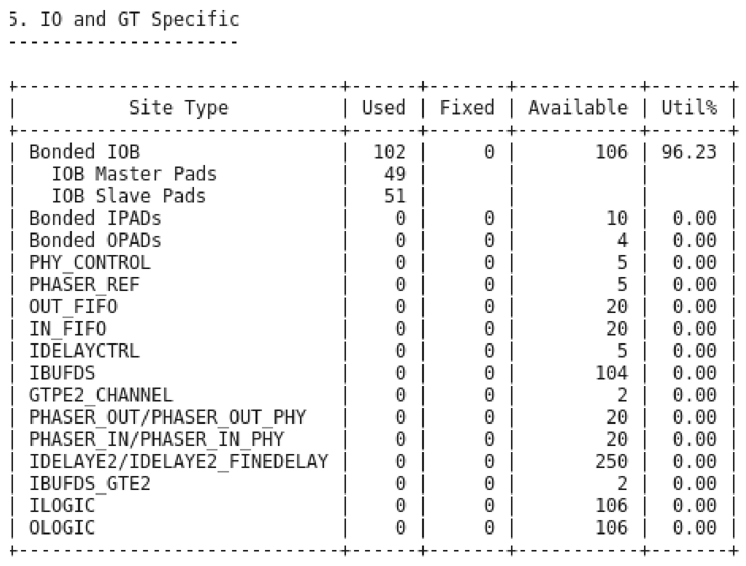
Figure 16 Divider Slice Logic

Because the I/O ports exceed the specifications, the synthesis and implementation results include the normalization block. The LUT used in divider is shown as the figure below.



Resource	Estimation	Available	Utilization %
LUT	801	20800	3.85
FF	281	41600	0.68
IO	129	106	121.70
BUFG	1	32	3.13

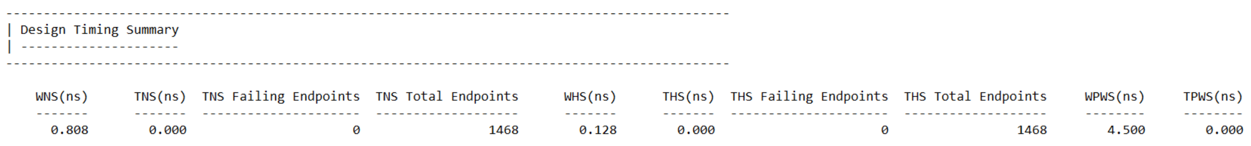
Figure 17 Divider Utilization



```
5. IO and GT Specific
--------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |  102 |     0 |       106 | 96.23 |
|   IOB Master Pads           |   49 |       |           |       |
|   IOB Slave Pads            |   51 |       |           |       |
| Bonded IPADs                |    0 |     0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |         5 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |       106 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

Figure 18 Divider IO

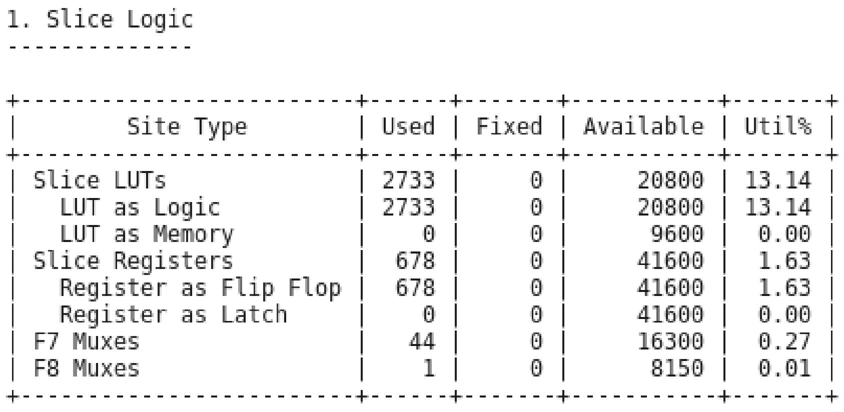


```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------  -------------------     --------     --------
      0.808        0.000                      0                 1468        0.128        0.000                      0                 1468        4.500        0.000
```

Figure 19 Divider Timing

7.3 uArch 3: Floating Point Multiplier (FP_mult.vhd)

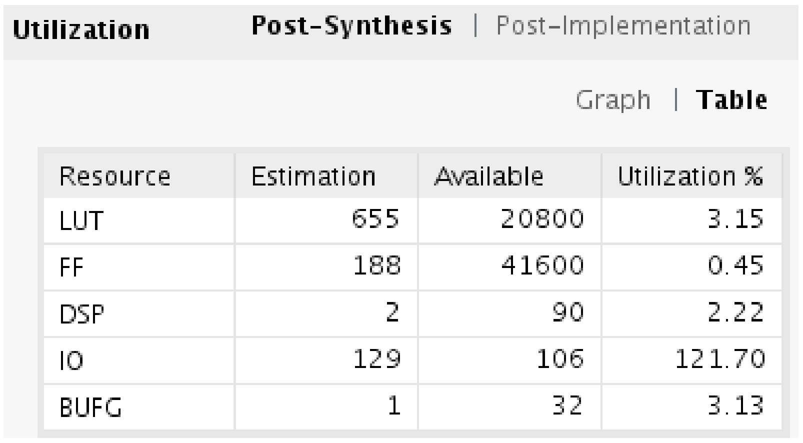


```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              | 2733 |     0 |     20800 | 13.14 |
|   LUT as Logic          | 2733 |     0 |     20800 | 13.14 |
|   LUT as Memory         |    0 |     0 |      9600 |  0.00 |
| Slice Registers         |  678 |     0 |     41600 |  1.63 |
|   Register as Flip Flop |  678 |     0 |     41600 |  1.63 |
|   Register as Latch     |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                |   44 |     0 |     16300 |  0.27 |
| F8 Muxes                |    1 |     0 |      8150 |  0.01 |
+-------------------------+------+-------+-----------+-------+
```

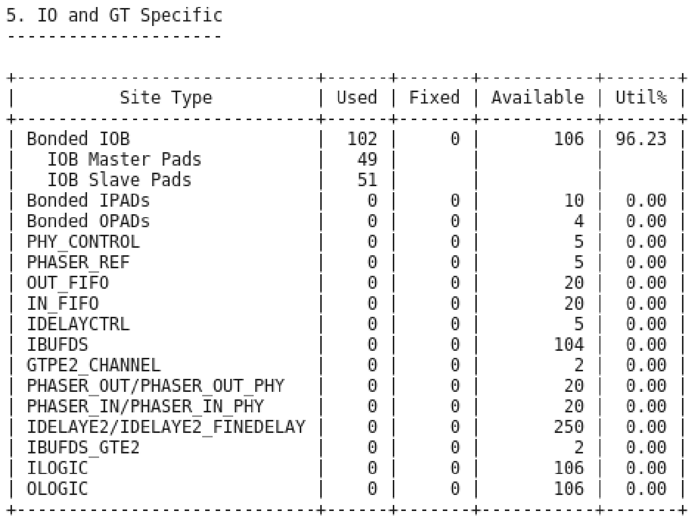
Figure 20 Multiplier Slice Logic

Because the I/O ports exceed the specifications, the synthesis and implementation results include the normalization block. The LUT used in multiplier is shown as the figure below.



Resource	Estimation	Available	Utilization %
LUT	655	20800	3.15
FF	188	41600	0.45
DSP	2	90	2.22
IO	129	106	121.70
BUFG	1	32	3.13

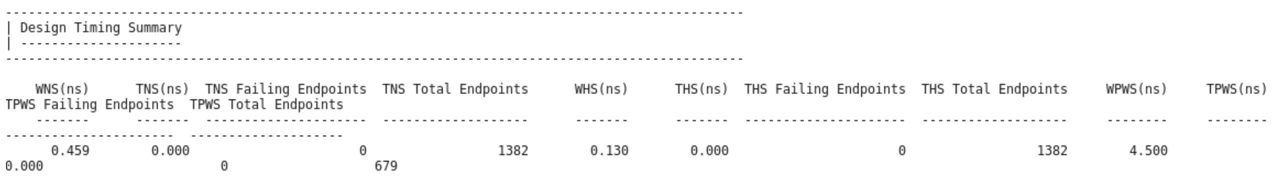
Figure 21 Multiplier Utilization



```
5. IO and GT Specific
---------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |  102 |     0 |       106 | 96.23 |
|   IOB Master Pads           |   49 |       |           |       |
|   IOB Slave Pads            |   51 |       |           |       |
| Bonded IPADs                |    0 |     0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |         5 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |       106 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

Figure 22 Multiplier IO

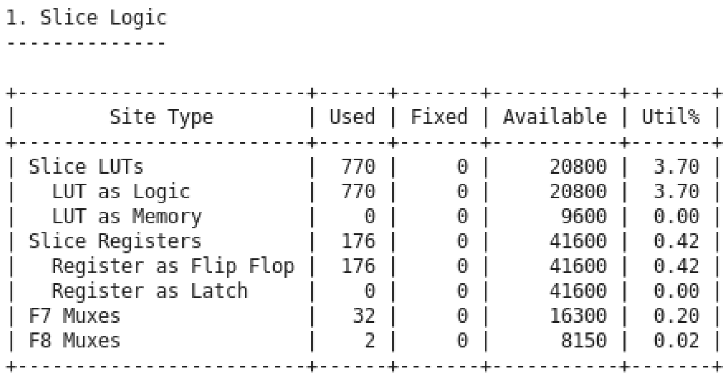


```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)
TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------  -------------------     --------     --------
----------------------  --------------------
      0.459        0.000                      0                 1382        0.130        0.000                      0                 1382        4.500
0.000                      0                   679
```

Figure 23 Multiplier Timing

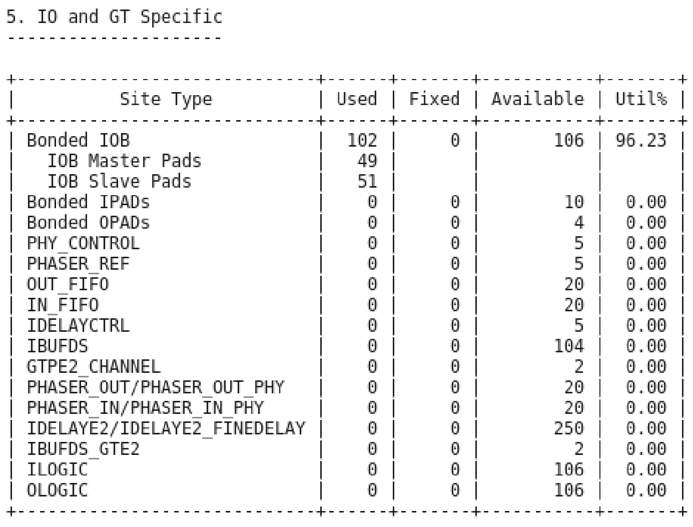
7.4 uArch 3: Normalization (Normalization.vhd)



```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              |  770 |     0 |     20800 |  3.70 |
|   LUT as Logic          |  770 |     0 |     20800 |  3.70 |
|   LUT as Memory         |    0 |     0 |      9600 |  0.00 |
| Slice Registers         |  176 |     0 |     41600 |  0.42 |
|   Register as Flip Flop |  176 |     0 |     41600 |  0.42 |
|   Register as Latch     |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                |   32 |     0 |     16300 |  0.20 |
| F8 Muxes                |    2 |     0 |      8150 |  0.02 |
+-------------------------+------+-------+-----------+-------+
```

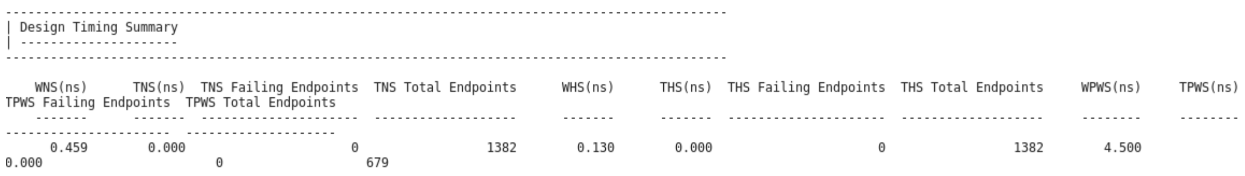
Figure 24 Multiplier Slice Logic



```
5. IO and GT Specific
--------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |  102 |     0 |       106 | 96.23 |
|   IOB Master Pads           |   49 |       |           |       |
|   IOB Slave Pads            |   51 |       |           |       |
| Bonded IPADs                |    0 |     0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |         5 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |       106 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

Figure 25 Multiplier IO

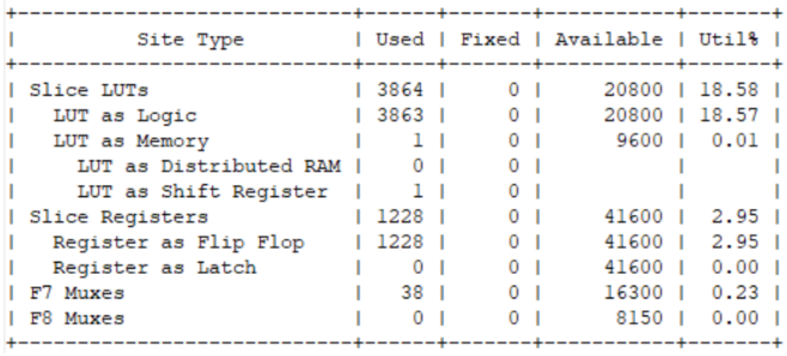


```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)
TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  --------------------  -------------------      -------      -------  ---------------------  -------------------     --------     --------
----------------------  --------------------
      0.459        0.000                     0                 1382        0.130        0.000                      0                 1382        4.500
0.000                          0                   679
```

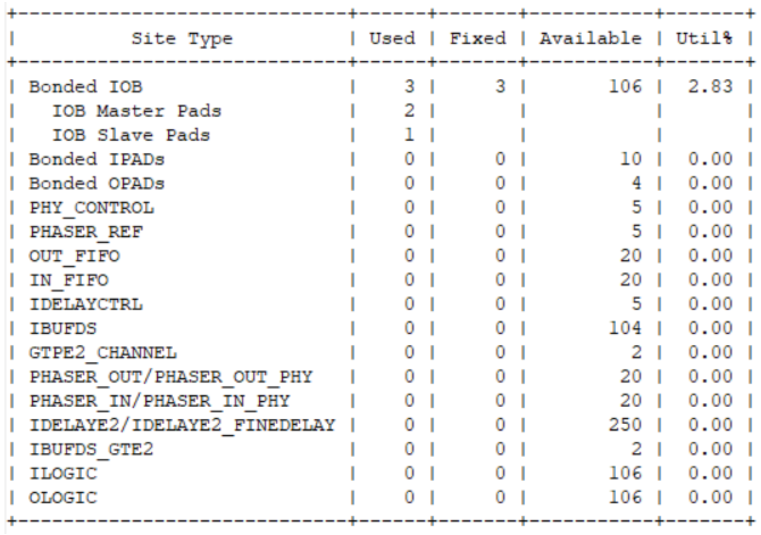
Figure 26 Multiplier Timing

7.5 System Level Implementation (TOP.vhd)



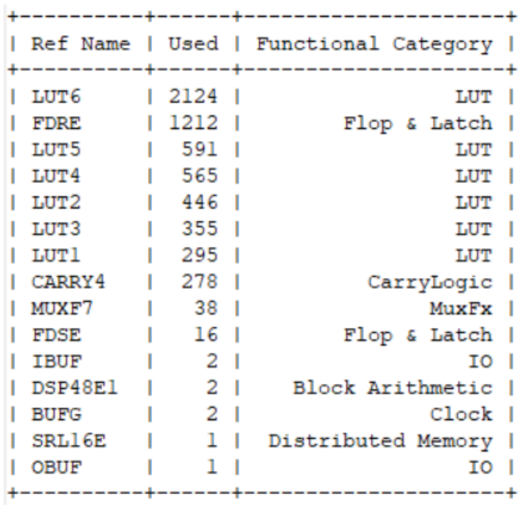
```
+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 3864 |     0 |     20800 | 18.58 |
|   LUT as Logic             | 3863 |     0 |     20800 | 18.57 |
|   LUT as Memory            |    1 |     0 |      9600 |  0.01 |
|     LUT as Distributed RAM |    0 |     0 |           |       |
|     LUT as Shift Register  |    1 |     0 |           |       |
| Slice Registers            | 1228 |     0 |     41600 |  2.95 |
|   Register as Flip Flop    | 1228 |     0 |     41600 |  2.95 |
|   Register as Latch        |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                   |   38 |     0 |     16300 |  0.23 |
| F8 Muxes                   |    0 |     0 |      8150 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Figure 27 system level slice logic



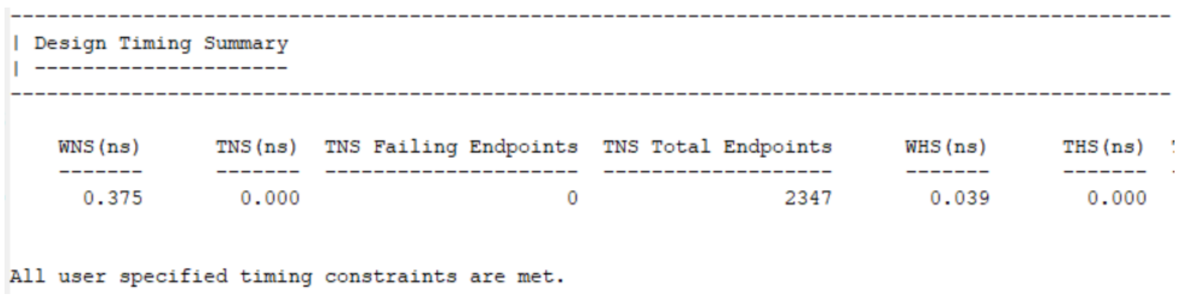
```
+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |    3 |     3 |       106 |  2.83 |
|   IOB Master Pads           |    2 |       |           |       |
|   IOB Slave Pads            |    1 |       |           |       |
| Bonded IPADs                |    0 |     0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |         5 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |       106 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

Figure 28 system level IO



```
+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| LUT6     | 2124 |                 LUT |
| FDRE     | 1212 |        Flop & Latch |
| LUT5     |  591 |                 LUT |
| LUT4     |  565 |                 LUT |
| LUT2     |  446 |                 LUT |
| LUT3     |  355 |                 LUT |
| LUT1     |  295 |                 LUT |
| CARRY4   |  278 |          CarryLogic |
| MUXF7    |   38 |               MuxFx |
| FDSE     |   16 |        Flop & Latch |
| IBUF     |    2 |                  IO |
| DSP48E1  |    2 |    Block Arithmetic |
| BUFG     |    2 |               Clock |
| SRL16E   |    1 |  Distributed Memory |
| OBUF     |    1 |                  IO |
+----------+------+---------------------+
```

Figure 29 system level primitives



```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
    -------      -------  ---------------------  -------------------      -------      -------
      0.375        0.000                      0                 2347        0.039        0.000


All user specified timing constraints are met.
```

Figure 30 system level timing

8 Work Distribution

- Overall architecture design (**Dingxin Jin**)
- Scripts & golden references
  Float(32bit) Reference_Generator.py (**Dingxin Jin**)
  Reference_Generator.vhd (**Dingxin Jin**)
  Reference_Generator_NoPipe.vhd (**Dingxin Jin**)
  tb_Reference_Generator.vhd (**Dingxin Jin**)
  tb_Reference_Generator_NoPipe.vhd (**Dingxin Jin**)
  Exception_Definition.xlsx (**Dingxin Jin**)
- Design files
  - uArch 1 UART
    UART_TX.vhd, UART_RX.vhd, UART.py (**Shengwei Lyu**)
    TOP.vhd (**Shengwei Lyu**) *Help debug and optimize (**Dingxin Jin**)
  - uArch 2 Floating Point Adder
    adder.vhd (**Peng Guo**) *Help debug and optimize (**Dingxin Jin**)
  - uArch 3 Floating Point Divider
    fpp_divide.vhd (**Xiaoting Lai**) *Help debug and optimize (**Dingxin Jin**)
  - uArch 4 Floating Point Multiplier
    FP_mult.vhd (**Shengwei Lyu**) *Help debug and optimize (**Dingxin Jin**)
  - uArch 5 Normalization
    Normalization.vhd (**Dingxin Jin**)
- Verification files
  tb_Multiplier_Wrapper.vhd (**Dingxin Jin**)
  tb_Wrapper.vhd(the same file for tb_adder.vhd) (**Dingxin Jin**)
- Solutions for Implementation battles
  - Battle 1 (**Xiaoting Lai**)
  - Battle 2 (**Peng Guo**)
  - Battle 3 (**Dingxin Jin**)
- System-level integration
  - Multiplier_Wrapper (**Dingxin Jin**)
  - Wrapper.vhd (**Dingxin Jin**)

9 Reference

[1] Muller, Jean-Michel, et al. Handbook of Floating-Point Arithmetic. Birkhäuser, 2018.
[2] Meyer-Baese, U. Digital Signal Processing with Field Programmable Gate Arrays. Springer, 2004.
[3] Shen, Ming-fa. "Implementation of Float Number Operation on CPLD/FPGA Chip by VHDL." Journal of Jinan University.
[4] "Tools & Thoughts." VisibleEarth High Resolution Map (43200x21600), www.h-schmidt.net/FloatConverter/IEEE754.html.
[5] Bishop, David. Floating point package user's guide, 2018.
[6] Rathor, Ajay, and Lalit Bandil. "Fpga Implementation of Floating - Point Arithmetic." International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE), vol. 1, no. 9, Nov. 2012, pp. 67–73.